# CSC1135 Secure Programming Lab 02

Darragh O'Brien B.Sc. Ph.D. FHEA
School of Computing
Dublin City University

## Overview

In this lab we will:

- Introduce `gdb`
- Explore process layout in memory
- Examine stack layout in memory
- Disassemble some C and annotate the resultant assembly
- Reverse engineer some C from assembly
- Rewrite a return address to control program execution

## A. Process layout in memory

Verify you are using `gcc` version 3.4:

```
$ gcc -v
```

Download a copy of program.c. Build an executable as follows:

```
$ gcc -g -o program program.c
```

Sketch an outline of the program's process address space. It should stretch from `0x00000000` at the bottom to `0xFFFFFFFF` at the top. Use `gdb` to locate (as best you can since it may not be possible to be precise in all cases) the following and mark them in your diagram:

- `.text` section start and end
- heap base address (approximately)
- `ptr` (where `ptr` points to)
- `&ptr` (where `ptr` lives)
- `argc` (where `argc` lives)
- `argv` (where `argv` lives)
- `argv[0]` (where `argv` points to)
- program entry point
- `.rodata` section start and end
- `I am a C program` (where the string lives)
- `big_array` (where `big_array` lives)
- `.data` section start and end
- stack start address (approximately)
- `buffer` (where `buffer` lives)
- `malloc` (where `malloc` is defined)
- `strcpy` (where `strcpy` is defined)
- `main` (where `main` is defined)

- `sumup` (where `sumup` is defined)
- `.bss` section start and end
- `swap` (where `swap` is defined)
- `local` (where `local` lives)

Here is an example gdb session which you might find useful in getting started. For extra help, here are some gdb notes, a gdb tutorial and a gdb quick reference. Or you could try the help available in `gdb` itself.

## B. Stack layout in memory

Set a breakpoint on the `swap` function in your program. Draw a diagram of the stack at the time the breakpoint is encountered. Your diagram should include arguments passed by the caller, the return address, the caller's frame pointer, the current frame pointer, local variables and any saved registers.

## C. Disassembling code

Disassemble the `swap` function and annotate each line with a description of what it does.

## D. Reverse engineering C from assembly

Translate each of the two assembly routines here into equivalent C functions. Each function returns an integer. Once you've made an attempt you can compare your answers to mine.

## E. Rewriting a return address

Rather than having control return to the `main` function, add C code to the `swap` function such that the `foo` function is executed when `swap` returns.

Hint: Declare a new pointer variable in `swap` and set it pointing to some other local variable; add an offset to it in order to have it point at the return address on the stack. Overwrite the return address with the address of the `foo` function. Buffer overflow attacks work in a similar way. Once you've made an attempt you can compare your approach to mine.